Department I - C Plus Plus

Modern and Lucid C++ for Professional Programmers

Week 5 – Errors, Classes and Operators

Prof. Peter Sommerlad / Thomas Corbat
Rapperswil, 15.10.2019
HS2019

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

Failing Functions

Goals:

- You know 5 different ways to react to errors in functions
- You know how to throw, catch and test exceptions

When Could a Function Fail?

- **Precondition (Assumption) is violated**
  - Negative index
  - Divisor is zero
  - Usually caller provided wrong arguments
- **A function without preconditions has a so-called "wide contract" as opposed to "narrow contract"**
- **Postcondition could not be satisfied**
  - Resources for computation are not available
  - Can not open a file

Contract cannot be fulfilled

- **What should you do, if a function cannot fulfill its purpose?**
  1. Ignore the error and provide potentially **undefined behavior**
  2. Return a **standard result** to cover the error
  3. Return an **error code** or error value
  4. Provide an **error status** as a side-effect
  5. Throw an **exception**
- **But first you need to know, if it can fail at all!**

```
std::vector v{1, 2, 3, 4, 5};
v[5] = 7;
```

- **Relies on the caller to satisfy all preconditions**
- **Viable only if not dependent on other resources**
- **Most efficient implementation**
  - No unnecessary checks
- **Simple for the implementer but harder for the caller**
- **Should be done consciously and consistently!**

```
std::string inputName(std::istream & in) {
  std::string name{};
  in >> name;
  return name.size() ? name : "anonymous";
}
```

- **Reliefs the caller from the need to care if it can continue with the default value**
- **Can hide underlying problems**
  - Debugging can give you nightmares
- **Often better if caller can specify its own default value**

```
std::string inputNameWithDefault(std::istream & in,
                                 std::string const & def = "anonymous") {
  std::string name{};
  in >> name;
  return name.size() ? name : def;
}
```

- **Only feasible if result domain is smaller than return type**
  - There exists a value that can be used
  - Sometimes invented artificially: `std::string::npos`

```
bool contains(std::string const & s, int number) {
  auto substring = std::to_string(number);
  return s.find(substring) != std::string::npos;
}
```

  - POSIX defines -1 to mark failure of many system calls
- **Burden on the caller to check the result**
  - Danger of ignoring significant errors if result is otherwise insignificant

#include <optional>

- **std::optional<T> extends the range of type T with an extra "no-value" value**
- **Encodes the possibility of failure in the type system**
  - Can optionally contain NO value (default construction)

```
std::optional<std::string> inputName(std::istream & in) {
  std::string name{};
  if (in >> name) return name;
  return {};
}
```

- **Requires explicit access of the value at the call site**
  - has_value() or boolean conversion checks whether the optional contains a value

```
int main() {
  std::optional name = inputName(std::cin);
  if (name.has_value()) {
    std::cout << "Name: " << name.value() << '\n';
  }
}
```

```
int main() {
  std::optional name = inputName(std::cin);
  if (name) {
    std::cout << "Name: " << *name << '\n';
  }
}
```

- **Requires reference parameter**
  - Can be `this` object in member functions
  - Annoying when error variable must be provided
- **(Bad!) Alternative: Global variable**
  - POSIX' **`errno`** is the glorious example of that

```
int connect(std::string url, bool & error) {
  //set error when an error occurred
}
```

- **Example: `std::istream`'s state (`good()`, `fail()`) is changed as a side-effect of input**

```
std::string name{};
in >> name;
if (in.fail()) { //Member variable
  //Handle error case
}
```

- **To throw an exception:**

```
throw std::invalid_argument{"Description"};
throw 15;
```

  - Any (copyable) type can be thrown
- **No means to specify what could be thrown**
  - No checks if you catch an exception that might be thrown at call-site
- **No meta-information is available as part of the exception**
  - No stack trace, no source position of throw
- **Exception thrown while exception is propagated results in program abort (not while caught)**
- **Use Exceptions for constructors that cannot guarantee class invariant and operator functions!**

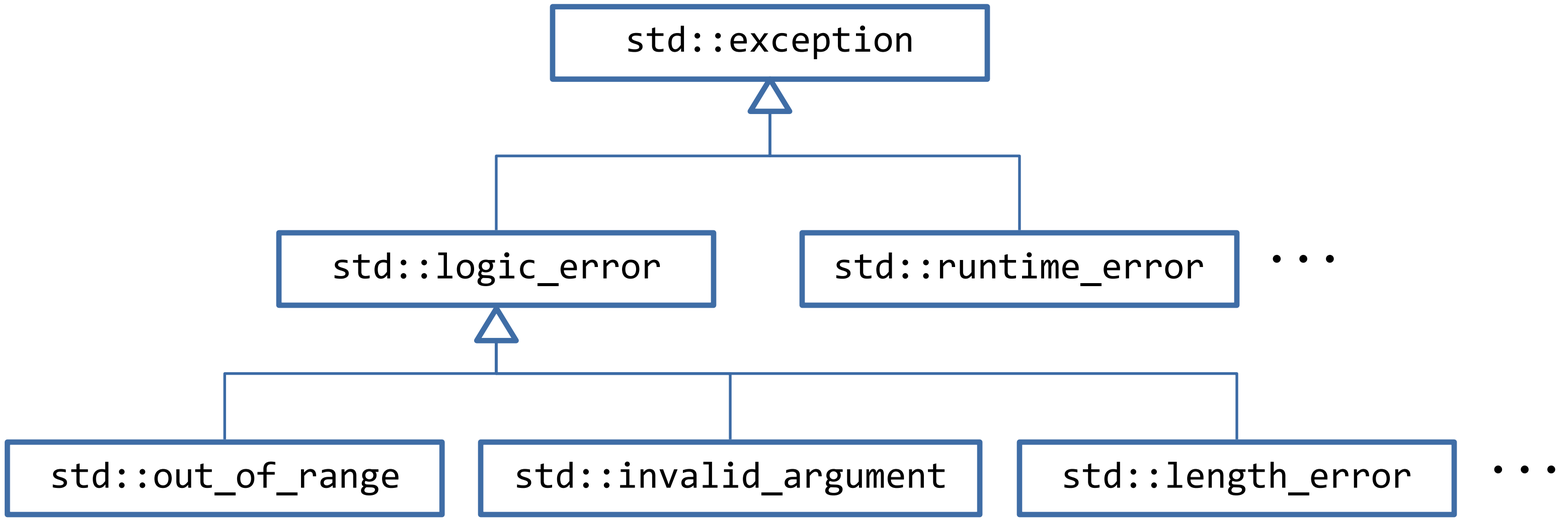
- **Try-catch block as in Java**
  - Can be the whole function body
- **Principle: Throw by value, catch by const reference**
  - Avoids unnecessary copying
  - Allows dynamic polymorphism for class types
- **Sequence of catches is significant**
  - First match wins
- **Catch all with ellipsis (...):**
  - Must be last catch
- **Caught exceptions can be rethrown with `throw;` in the catch block.**

```
try {
  throwingCall();
} catch (type const & e) {
  //Handle type exception
} catch (type2 const & e) {
  //Handle type2 exception
} catch (...) {
  //Handle other exception types
}
```

Predefined Exception Types

```
#include <stdexcept>
```


- **The Standard Library has some pre-defined exception types that you can also use in <stdexcept>**



- **All subclasses of `logic_error` and `runtime_error` have a constructor parameter for the "reason" of type `std::string`**

- **`std::exception` is the base class**

  - It provides the `what()` member function to obtain the "reason"

- **Functions that have a precondition on their caller**
  - When not all possible argument values are useful for the function

```
double square_root(double x) {
  if (x < 0) {
    throw std::invalid_argument{"square_root imaginary"};
  }
  return std::sqrt(x);
}
```

- **Do NOT use exceptions as a means to return values**
  - Catch then becomes a "come from" and throw a "go to"

- **CUTE provides `ASSERT_THROWS(code, exception)`**

```
void testSquareRootNegativeThrows() {
  ASSERT_THROWS(square_root(-1.0), std::invalid_argument);
}
```

```
void testEmptyVectorAtThrows() {
  std::vector<int> empty_vector{};
  ASSERT_THROWS(empty_vector.at(0), std::out_of_range);
}
```

- **You can also use `try-FAILM()-catch` (this is how `ASSERT_THROWS` is implemented)**

```
void testForExceptionTryCatch() {
  std::vector<int> empty_vector{};
  try {
    empty_vector.at(1);
    FAILM("expected Exception");
  } catch (std::out_of_range const &) {
    // expected
  }
}
```

Right or Wrong

X

?

```
void check(int i) {
  if (i % 2) {
    throw "is even";
  throw 0;
}

void printIsEven(int i) try {
  check(i);
} catch(int) {
  std::cout << "that's odd";
} catch(...) {
  std::cout << "very even";
}
```

Right or Wrong

X

?

```
void check(int i) {
  if (i % 2) {
    throw "is even";
  throw 0;
}

void printIsEven(int i) try {
  check(i);
} catch(int) {
  std::cout << "that's odd";
} catch(...) {
  std::cout << "very even";
}
```

https://cdn.someecards.com/someecards/usercards/that-is-just-so-wrong-on-so-many-levels-19e73.png

Right or Wrong

X

?

```
void check(int i) {
  if (i % 2) {
    throw "is even";
  throw 0;
}

void printIsEven(int i) try {
  check(i);
} catch(int) {
  std::cout << "that's odd";
} catch(...) {
  std::cout << "very even";
}
```

Right or Wrong

X

?

```
void check(int i) {
  if (i % 2) {
    throw "is even";
  throw 0;
}

void printIsEven(int i) try {
  check(i);
} catch(int) {
  std::cout << "that's odd";
} catch(...) {
  std::cout << "very even";
}
```

Incorrect

It is syntactically correct C++, BUT:

- You must never use an exception on correct execution paths just to change control flow on the call-site. Use exceptions only for violations of pre- and post-conditions!
- Don't throw primitives, even if you can. Use an exception from <stdexcept> or derive your own exception from std::exception.
- Catch by const reference, not by value!
- Condition is inverted, so the exception is lying

Taxonomy of errors (adopted from Herb Sutter Keynote Cppcon 2019)

https://www.youtube.com/watch?v=ARYP83yNAWk

Taxonomy of errors (adopted from Herb Sutter Keynote Cppcon 2019)

Conceptual goals for error free and fault tolerant software	What to use	Report-to Handler	Handler species

https://www.youtube.com/watch?v=ARYP83yNAWk

Taxonomy of errors (adopted from Herb Sutter Keynote Cppcon 2019)

Conceptual goals for error free and fault tolerant software	What to use	Report-to Handler	Handler species
Corruption of the abstract machine (e.g., no more stack,UB)	std::terminate()	User	Human

https://www.youtube.com/watch?v=ARYP83yNAWk

Taxonomy of errors (adopted from Herb Sutter Keynote Cppcon 2019)

Conceptual goals for error free and fault tolerant software	What to use	Report-to Handler	Handler species
Corruption of the abstract machine (e.g., no more stack,UB)	std::terminate()	User	Human
Programming bug - detectable (e.g., precondition violation)	asserts, log checks, contracts, ...	Programmer	Human write better unit tests !

https://www.youtube.com/watch?v=ARYP83yNAWk

Taxonomy of errors (adopted from Herb Sutter Keynote Cppcon 2019)

Conceptual goals for error free and fault tolerant software	What to use	Report-to Handler	Handler species
Corruption of the abstract machine (e.g., no more stack,UB)	std::terminate()	User	Human
Programming bug - detectable (e.g., precondition violation)	asserts, log checks, contracts, ...	Programmer	Human write better unit tests !
Recoverable/expected error (e.g., host not found)	throw exception, error code, etc.	Calling Code	Code for increasing fault tolerance

https://www.youtube.com/watch?v=ARYP83yNAWk

- **A good function**
  - Does one thing well and is named after that ("High Cohesion")
  - Has only few parameters (<=3, max 5)
  - Consists of only a few lines without deeply nested control structure
  - Provides guarantees about its result (aka its contract)
  - Is easy to use with all possible argument values its parameter types allow or provides consistent error reporting if argument values prohibit delivering its result (exception)
- **Pass parameters and return results by value (unless there is a good reason not to)**

- Functions can be declared to explicitly not throw an exception with the noexcept keyword
- The compiler does not need to check it

```
int add(int lhs, int rhs) noexcept
{
  return lhs + rhs;
}
```

- If an exception is thrown (directly or indirectly) from a noexcept function the program will terminate

```
void fail() {
  throw 1;
}

void lie() noexcept {
  fail();
}
```

Classes

Goals:

- You know how to define a class in C++
- You know the elements a class consists of
- You can implement your own data types

A Good Class

- **Does one thing well and is named after that**
  - High Cohesion
- **Consists of member functions with only a few lines**
  - Avoid deeply nested control structures
- **Has a class invariant and guarantees it if needed**
  - Provides a guarantee about its state (values of the member variables)
  - Constructors establish that invariant
- **Is easy to use without complicated protocol sequence requirements**

GoodClassName.h

```
class <GoodClassName> {

  <constructors>

  <member functions>

  <member variables>

};
```

- **A class defines a new type**
- **A class is usually defined in a header file**
- **At the end of a class definition a semicolon is required**

Date.h

```
#ifndef DATE_H_
#define DATE_H_

class Date {
public:
  Date(int year, int month, int day)
  : year{year}, month{month}, day{day} {/*...*/}

  static bool isLeapYear(int year) {/*...*/}

private:
  bool isValidDate() const {/*...*/}
  int year, month, day;
};

#endif /* DATE_H_ */
```

- **Include guard ensures that the content of a header file is only included once**
  - Eliminates cyclic dependencies of `#include` directives
- **Prevents violation of the One Definition Rule**
- **Directives**
  - `#ifndef <name>`
  - `#define <name>`
  - `#endif`
  - `#pragma once // non standard!`

Date.h

```
#ifndef DATE_H_
#define DATE_H_

class Date {
public:
  Date(int year, int month, int day)
  : year{year}, month{month}, day{day} {/*...*/}

  static bool isLeapYear(int year) {/*...*/}

private:
  bool isValidDate() const {/*...*/}
  int year, month, day;
};

#endif /* DATE_H_ */
```

- **Keywords for defining a class**
  - `class`
  - `struct`
- **Default visibility for members of the class are**
  - `private` for `class`
  - `public` for `struct`

```
struct <name> {
  ...
};
```

Date.h

```
#ifndef DATE_H_
#define DATE_H_

class Date {
public:
  Date(int year, int month, int day)
  : year{year}, month{month}, day{day} {/*...*/}

  static bool isLeapYear(int year) {/*...*/}

private:
  bool isValidDate() const {/*...*/}
  int year, month, day;
};

#endif /* DATE_H_ */
```

- **Access specifiers (followed by a colon :)**
  - `private:`
    visible only inside the class (and friends); for hidden data members
  - `protected:`
    also visible in subclasses
  - `public:`
    visible from everywhere; for the interface of the class
- **All subsequent members have this visibility**
- **Each visibility can reoccur multiple times**

Date.h

```
#ifndef DATE_H_
#define DATE_H_

class Date {
public:
  Date(int year, int month, int day)
  : year{year}, month{month}, day{day} {/*...*/}

  static bool isLeapYear(int year) {/*...*/}

private:
  bool isValidDate() const {/*...*/}
  int year, month, day;
};

#endif /* DATE_H_ */
```

- **Have a type and a name**

```
<type> <name>;
```

- **The content/state of an object that represents the value of the type**
- **Make members `const` if possible**
- **Don't add members to communicate between member function calls**
  - Hard to test
  - Such usage protocols are a burden for the user of the class
  - Better: Use parameters

Date.h

```
#ifndef DATE_H_
#define DATE_H_

class Date {
public:
  Date(int year, int month, int day)
  : year{year}, month{month}, day{day} {/*...*/}

  static bool isLeapYear(int year) {/*...*/}

private:
  bool isValidDate() const {/*...*/}
  int year, month, day;
};

#endif /* DATE_H_ */
```

- **Function with name of the class**
  - Special member function
- **No return type**

```
<class name>(){}
```

- **Initializer list for member initialization**

```
<class name>(<parameters>)
  : <initializer-list>
{}
```

Date.h

```
#ifndef DATE_H_
#define DATE_H_

class Date {
public:
  Date(int year, int month, int day)
  : year{year}, month{month}, day{day} {/*...*/}

  static bool isLeapYear(int year) {/*...*/}

private:
  bool isValidDate() const {/*...*/}
  int year, month, day;
};

#endif /* DATE_H_ */
```

- **Function with name of the class**
  - Special member function
- **No return type**

```
<class name>(){}
```

- **Initializer list for member initialization**

```
<class name>(<parameters>)
  : <initializer-list>
{}
```

Date.h

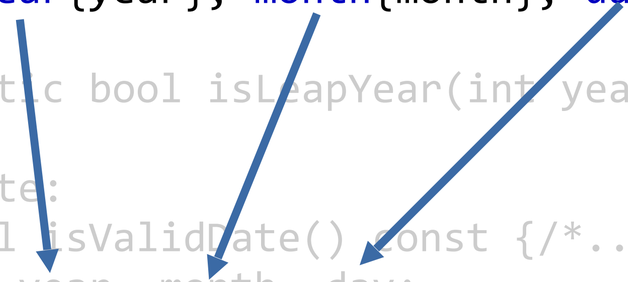
```
#ifndef DATE_H_
#define DATE_H_

class Date {
public:
  Date(int year, int month, int day)
  : year{year}, month{month}, day{day} {/*...*/}

  static bool isLeapYear(int year) {/*...*/}

private:
  bool isValidDate() const {/*...*/}
  int year, month, day;
};

#endif /* DATE_H_ */
```



- **Default Constructor**

```
Date d{};
```

  - No parameters
  - Implicitly available if there are no other explicit constructors
  - Has to initialize member variables with default values

```
Date d2{d};
```

- **Copy Constructor**
  - Has one `<own-type> const &` parameter
  - Implicitly available (unless there is an explicit move constructor or assignment operator)
  - Copies all member variables
  - Usually you don't need to define or implement it

```
class Date {
public:
  Date(int year, int month, int day);
  //Default-Constructor
  Date();
  //Copy-Constructor
  Date(Date const &);
  //Move-Constructor
  Date(Date &&);
  //Typeconversion-Constructor
  explicit Date(std::string const &);
  //Destructor
  ~Date();
};
```

- **Move Constructor**

```
Date d2{std::move(d)};
```

  - Has one `<own-type>` && parameter
  - Implicitly available (unless there is an explicit copy constructor or assignment operator)
  - Moves all members
  - Usually you don't need to implement it explicitly

- **Will be covered in C++ Advanced**

```
class Date {
public:
  Date(int year, int month, int day);
  //Default-Constructor
  Date();
  //Copy-Constructor
  Date(Date const &);
  //Move-Constructor
  Date(Date &&);
  //Typeconversion-Constructor
  explicit Date(std::string const &);
  //Destructor
  ~Date();
};
```

- **Typeconversion Constructor**

```
Date tomorrow{"16/10/2019"s};
```

  - Has one `<other-type> const &` parameter
  - Converts the input type if possible
  - Declare `explicit` to avoid unexpected conversions!

```
class Date {
public:
  Date(int year, int month, int day);
  //Default-Constructor
  Date();
  //Copy-Constructor
  Date(Date const &);
  //Move-Constructor
  Date(Date &&);
  //Typeconversion-Constructor
  explicit Date(std::string const &);
  //Destructor
  ~Date();
};
```

- **Initializer List Constructor**

```
Container box{item1, item2, item3};
```

  - Has one `std::initializer_list` parameter
  - Does not need to be `explicit`, implicit conversion is usually desired

```
struct Container {
  Ccontainer() = default;
  Container(std::initializer_list<Element> elements);
private:
  std::vector<Element> elements{};
};
```

- **Initializer List constructors are preferred if a variable is initialized with {}**

```
std::vector v(5, 10)
```

```
std::vector v{5, 10}
```

- **Named like the default constructor but with a leading ~**

```
~Date();
```

- **Has to release all resources**
- **Implicitly available**
  - If you program properly you will hardly ever need to implement it yourself!
- **Must not throw an exception!**
- **Called automatically at the end of the block for local instances (} destroys all locals)**

```
class Date {
public:
  Date(int year, int month, int day);
  //Default-Constructor
  Date();
  //Copy-Constructor
  Date(Date const &);
  //Move-Constructor
  Date(Date &&);
  //Typeconversion-Constructor
  explicit Date(std::string const &);
  //Destructor
  ~Date();
};
```

- **Base classes are specified after the name**

```
class <name> : <base1>, ..., <baseN>
```

- **Multiple inheritance is possible**
- **Inheritance can specify a visibility**
  - `public, protected, private`
  - Limits the **maximum** visibility of the inherited members
  - If no visibility is specified the default of the inheriting class is used (`class->private` and `struct->public`)
- **Details about Diamonds and Virtual inheritance later**

```
class Base {
private:
  int onlyInBase;
protected:
  int baseAndInSubclasses;
public:
  int everyoneCanFiddleWithMe
};
```

```
class Sub : public Base {
  //Can see baseAndInSubclasses and
  //everyoneCanFiddleWithMe
};
```

Separation of Declaration and Implementation (Cevelop: Toggle refactoring)

Date.h

```
#ifndef DATE_H_
#define DATE_H_

class Date {
  int year, month, day;
public:
  Date(int year, int month, int day);

  static bool isLeapYear(int year);

private:
  bool isValidDate() const;
};

#endif /* DATE_H_ */
```

Date.cpp

```
#include "Date.h"

Date::Date(int year, int month, int day)
  : year{year}, month{month}, day{day} {
  /*...*/
}

bool Date::isLeapYear(int year) {
  /*...*/
}

bool Date::isValidDate() const {
  /*...*/
}
```

Using the Class

Any.cpp

```
#include "Date.h"

void foo() {
  Date today{2019, 10, 15};

  auto wednesday{today.tomorrow()};

  Date::isLeapYear(2016);

  //what now?
  Date invalidDate {2019, 13, 1};
}
```

Date.h

```
#ifndef DATE_H_
#define DATE_H_

class Date {
  int year, month, day;
public:
  Date(int year, int month, int day);

  Date tomorrow() const;

  static bool isLeapYear(int year);

private:
  bool isValidDate() const;
};

#endif /* DATE_H_ */
```

Right or Wrong?

X


```
struct Recipe {
  Recipe(std::initializer_list<Step> steps);
  Meal cook() const;
private:
  std::vector<Step> steps{};
};
```

```
struct Vehicle {
  Location location{};
};
class Car : Vehicle {
public:
  Route drive(Destination destination);
};
void printLocation(Car & car) {
  std::cout << car.location;
}
```

Right or Wrong?

X

?

```
struct Recipe {
  Recipe(std::initializer_list<Step> steps);
  Meal cook() const;
private:
  std::vector<Step> steps{};
};
```

Correct

The class declaration has a semicolon at the end and a Recipe is constructible with a list of (cooking) steps. The visibilities are sensible.

```
struct Vehicle {
  Location location{};
};
class Car : Vehicle {
public:
  Route drive(Destination destination);
};
void printLocation(Car & car) {
  std::cout << car.location;
}
```

Right or Wrong?


```
struct Recipe {
  Recipe(std::initializer_list<Step> steps);
  Meal cook() const;
private:
  std::vector<Step> steps{};
};
```

Correct

The class declaration has a semicolon at the end and a Recipe is constructible with a list of (cooking) steps. The visibilities are sensible.

```
struct Vehicle {
  Location location{};
};
class Car : Vehicle {
public:
  Route drive(Destination destination);
};
void printLocation(Car & car) {
  std::cout << car.location;
}
```

Inncorrect

While `Vehicle` and `Car` are syntactically correct, `Car` inherits privately from `Vehicle` (due to the class keyword). Therefore, the members of `Vehicle` cannot be accessed from outside the `Car` class.

- **Establish Invariant**
  - Properties for a value of the type that are always true
  - E.g. a `Date` instance always represents a valid date
  - All (public) member functions assume and keep it intact
- **Initialize all members**
  - Constructors only create a valid instance (Otherwise throw an exception!)
  - Use list of base classes and member variables
  - Use default values if possible/necessary

Date.cpp

```
#include "Date.h"

Date::Date(int year, int month, int day)
  : year{year}, month{month}, day{day} {
  if (!isValidDate()) {
    throw std::out_of_range{"invalid date"};
  }
}

Date::Date() : Date{1980, 1, 1} {
}

Date::Date(Date const & other)
  : Date{other.year, other.month, other.day} {
}
```

- **As we have specified a default value, this should be the value created by the default constructor**
  - Constructor without parameters
- **Remark regarding initialization:**

```
Date nice_d{};
Date ugly_d;
```

  - Both create a Date and call the default constructor in this case
  - The second (without {}) does not work with all types. It might contain uninitialized variables
  - Good practice: Initialize all variables with {}!

Date.h

```
#ifndef DATE_H_
#define DATE_H_

class Date {
  //...
  Date();
};

#endif /* DATE_H_ */
```

Date.cpp

```
#include "Date.h"

Date::Date()
  : year{9999}, month{12}, day{31} {
}
```

- **Member variables can have a default value assigned**
  - NSDMI – Non-Static Data Member Initializers

```
class <classname> {
  <type> <membername>{<default-value>};
};
```

- **Such values are used if the member is not present in the initializer list of the constructor**
  - Initializer list still overrides those values
- **Useful if multiple constructors initialize date similarly**
  - Avoids duplication

Date.h

```
#ifndef DATE_H_
#define DATE_H_

class Date {
  int year{9999}, month{12}, day{31};
  //...
  Date();
  Date(int year, int month, int day);
};

#endif /* DATE_H_ */
```

Date.cpp

```
#include "Date.h"

Date::Date() {
}

Date::Date(int year, int month, int day)
  : year{year}, month{month}, day{day} {
  /*...*/
}
```

- **Some special member functions are implicitly available in certain cases**
  - E.g. Default constructor is implicitly available if no other explicit constructor is declared
- **(Re-)implementing the default behavior of the default constructor can be avoided:**

```
<ctor-name>() = default;
```

  - Adds the corresponding constructor to the type with the same behavior as if it was implicitly available
  - Possible for:
    Default constructor and destructor
    Copy/move constructor
    Copy/move assignment operator

Date.h

```
#ifndef DATE_H_
#define DATE_H_

class Date {
  int year{9999}, month{12}, day{31};
  //...
  Date() = default;
  Date(int year, int month, int day);
};

#endif /* DATE_H_ */
```

Date.cpp

```
#include "Date.h"

Date::Date() {
}

Date::Date(int year, int month, int day)
  : year{year}, month{month}, day{day} {
  /*...*/
}
```

Deleted Constructors

- **Some special member functions are implicitly available in certain cases**
  - E.g. Default constructor is implicitly available if no other explicit constructor is declared
- **Those implicit constructor are not always wanted**
  - Make them explicitly private
  - Or delete them

```
<ctor-name>() = delete;
```

  - Possible for:
    Default constructor and destructor
    Copy/move constructor
    Copy/move assignment operator

Banknote.h

```
#ifndef BANKNOTE_H_
#define BANKNOTE_H_

class Banknote {
  int value;
  //...
  Banknote(Banknote & const) = delete;
};

#endif /* BANKNOTE_H_ */
```

Forger.cpp

```
#include "Banknote.h"

Banknote forge(Banknote const & note) {
  Banknote copy{note}; //!
  return copy;
}
```

Delegating Constructors

- **Similar to Java constructors can call other constructors**

C++

```
Ctor(Parameters)
  : Ctor(Arguments) {
}
```

Java

```
Ctor(Parameters) {
  this(Arguments);
}
```

- **Constructor call has to be in the member initializer list**

- **Similar are calls to constructors of base classes**

C++

```
Ctor(Parameters)
  : Base(Arguments) {
}
```

Java

```
Ctor(Parameters) {
  super(Arguments);
}
```

Date.h

```
#ifndef DATE_H_
#define DATE_H_

class Date {
  //...
  Date(int year, Month month, int day);
  Date(int year, int month, int day);
};

#endif /* DATE_H_ */
```

Date.cpp

```
#include "Date.h"

Date::Date(int year, int month, int day)
      : Date{year, Month(month), day} {}
```

- **Don’t violate invariant**
  - Leave object in valid state
- **Implicit `this` object**
  - Is a pointer
  - Member access with arrow: `->`
- **Declare `const` if possible!**
- **Must not modify members if `const`**
  - Refers to `this` object
  - Can only call `const` members
- **Otherwise access to all other members**

Date.cpp

```
#include "Date.h"

bool Date::isValidDate() const {
  if (day <= 0) {
    return false;
  }
  switch (month) {
    case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
      return day <= 31;
    case 4: case 6: case 9: case 11:
      return this->day <= 30;
    case 2:
      return day <= (isLeapYear(year) ? 29:28);
    default:
      return false;
  }
}
```

Right or Wrong?


```
struct Counter {
  void increase(unsigned step) {
    auto before = current();
    value = before + step;
  }
  unsigned current() const;
private:
  unsigned value;
};
```

```
struct Document {
  void print(std::ostream & out) const {
    updatePrintDate();
    out << content;
  }
private:
  void updatePrintDate();
};
```

Right or Wrong?


```
struct Counter {
  void increase(unsigned step) {
    auto before = current();
    value = before + step;
  }
  unsigned current() const;
private:
  unsigned value;
};
```

Correct

It is allowed to call const member functions from non-const member functions.

```
struct Document {
  void print(std::ostream & out) const {
    updatePrintDate();
    out << content;
  }
private:
  void updatePrintDate();
};
```

Right or Wrong?


```
struct Counter {
  void increase(unsigned step) {
    auto before = current();
    value = before + step;
  }
  unsigned current() const;
private:
  unsigned value;
};
```

Correct

It is allowed to call const member functions from non-const member functions.

```
struct Document {
  void print(std::ostream & out) const {
    updatePrintDate();
    out << content;
  }
private:
  void updatePrintDate();
};
```

Inncorrect

It is not allowed to call a non-const member function from a const member function

- **No `this` object**
- **Cannot be `const`**
- **No `static` keyword in implementation**
- **Call with *<classname>*::*<member>*()**

```
Date::isLeapYear(2016);
```

Date.cpp

```
#include "Date.h"

bool Date::isLeapYear(int year) {
  if (year % 400 == 0) {
    return true;
  } else if (year % 100 == 0) {
    return false;
  } else if (year % 4 == 0) {
    return true;
  }
  return false;
}

//or the unreadable version
bool Date::isLeapYear(int year) {
  return !(year % 4) &&
         ((year % 100) || !(year % 400));
}
```

Static Member Variables - caution: these are globals in disguise

- **No `static` keyword in implementation (should not be used, because of threading)**
- **`static const/inline` member can be initialized directly (no ODR violation)**
- **Access outside class with name qualifier: *`<classname>::<member>`***

Date.h

```
class Date {
  static const Date myBirthday;
  static Date favoriteStudentsBirthday;
  static const Date today{2018, 10, 16};

  //...
};
```

Any.cpp

```
#include "Date.h"
#include <iostream>

void foo() {
  std::cout << Date::myBirthday;
  Date::favoriteStudentsBirthday = …;
}
```

Date.cpp

```
#include "Date.h"


Date const Date::myBirthday{1964, 12, 24};

Date Date::favoriteStudentsBirthday{1995, 5, 10};
```

Right or Wrong?


```
struct Recipe {
  Recipe(std::vector<Step> steps) = default;
  Meal cook() const;
private:
  std::vector<Step> steps{};
};
```

```
struct Chair {
  explicit Chair(unsigned legs = 4u);
private:
  unsigned legs;
};
Chair::Chair(unsigned legs)
  : legs{legs} {
  if (legs < 1u) {
    throw std::invalid_argument{"..."};
  }
}
```

Right or Wrong?


```
struct Recipe {
  Recipe(std::vector<Step> steps) = default;
  Meal cook() const;
private:
  std::vector<Step> steps{};
};
```

Incorrect

Of all constructors, only default, copy and move constructors can be declared = `default`.

```
struct Chair {
  explicit Chair(unsigned legs = 4u);
private:
  unsigned legs;
};
Chair::Chair(unsigned legs)
  : legs{legs} {
  if (legs < 1u) {
    throw std::invalid_argument{"..."};
  }
}
```

Right or Wrong?

?

```
struct Recipe {
  Recipe(std::vector<Step> steps) = default;
  Meal cook() const;
private:
  std::vector<Step> steps{};
};
```

Incorrect

Of all constructors, only default, copy and move constructors can be declared = `default`.

```
struct Chair {
  explicit Chair(unsigned legs = 4u);
private:
  unsigned legs;
};
Chair::Chair(unsigned legs)
  : legs{legs} {
  if (legs < 1u) {
    throw std::invalid_argument{"..."};
  }
}
```

Correct

Constructors can have default arguments too. In the declaration, a constructor that can be called with a single argument should be explicit. It uses the member initializer list and throws an exception if the invariant cannot be established.

Operator Overloading

Goals:

- You know how to overload operators for classes
- You know the correct way to read and print objects
- You can deal with streams correctly in your classes

- **Custom operators can be overloaded for user-defined types**
- **Declared like a function, with a special name**

```
<returntype> operator op(<parameters>);
```

  - Unary operators -> one parameter
  - Binary operators -> two parameters
- **Implement operators reasonably!**
  - Semantic should be natural
- **"When in doubt, do as the `int`s do"**
  - Scott Meyers – Effective C++

- **Overloadable Operators (*op*):**

```
+     -     *     /        %        ^
&     |     ~     !        ,        =
<     >     <=    >=       ++       --
<<    >>    ==    !=       &&       ||
+=    -=    /=    %=       ^=       &=
|=    *=    <<=   >>=      []       ()
->    ->*   new   new []   delete   delete []
```

- **Non-Overloadable Operators:**

```
::          .*          .          ?:
```

- **Example: Making `Date` comparable**
- **Compare `year`, `month` and `day`**
- **Free `operator<(l,r)`**
  - Two parameters of type **`Date`**
  - Each **`const &`**
  - Return type **`bool`**
- **`inline` when defined in header**
- **Problem with free operator**
  - No access to private members

Any.cpp

```
#include "Date.h"
#include <iostream>

void foo() {
  std::cout << Date::myBirthday;
  Date d{};
  std::cin >> d;
  std::cout << "is d older? " << (d < Date::myBirthday);
}
```

Date.h

```
class Date {
  int year, month, day; //private ☹
};

inline bool operator<(Date const & lhs, Date const & rhs) {
  return lhs.year < rhs.year ||
    (lhs.year == rhs.year && (lhs.month < rhs.month ||
      (lhs.month == rhs.month && lhs.day == rhs.day)));
}
```

- **Member `operator<`**
  - One parameters of type **`Date`**
  - Which is **`const &`**
  - Return type **`bool`**
  - Right-hand side of operation
- **Implicit `this` object**
  - `const` due to qualifier
  - Left-hand side of operation
- **Access to private members**
- **Implicit `inline` as member**

Any.cpp

```cpp
#include "Date.h"
#include <iostream>

void foo() {
  std::cout << Date::myBirthday;
  Date d{};
  std::cin >> d;
  std::cout << "is d older? " << (d < Date::myBirthday);
}
```

Date.h

```cpp
class Date {
  int year, month, day; //private ☺️

  bool operator<(Date const & rhs) const {
    return year < rhs.year ||
      (year == rhs.year && (month < rhs.month ||
        (month == rhs.month && day == rhs.day)));
  }
};
```

- **`std::tie` creates a tuple and binds the arguments with lvalue references**
- **std::tuple provides comparison operators:**
  - `operator==`
    `operator!=`
    `operator<`
    `operator<=`
    `operator>`
    `operator>=`
- **Comparison of `std::tuple` is component-wise from left to right**

C++20 introduces <=> spaceship operator for comparisons reducing the boilerplate to write

Date.cpp

```cpp
#include "Date.h"
#include <tuple>

bool Date::operator<(Date const & rhs) const {
  return std::tie(year, month, day) <
         std::tie(rhs.year, rhs.month, rhs.day);
}
```

Date.h

```cpp
class Date {
  int year, month, day; //private ☺️

  bool operator<(Date const & rhs) const;
};
```

Implementing All Comparisons

- **Ensure**
  - Transitivity
  - Associativity
  - Commutativity
- **Avoid duplication!**
- **Beware of call loops!**
- **Caution: can slow compile times**
- **Better: friend inline within class**

Date.h

```
class Date {
  int year, month, day; //private ☺️
public:
  bool operator<(Date const & rhs) const;
};

inline bool operator>(Date const & lhs, Date const & rhs) {
  return rhs < lhs;
}
inline bool operator>=(Date const & lhs, Date const & rhs) {
  return !(lhs < rhs);
}
inline bool operator<=(Date const & lhs, Date const & rhs) {
  return !(rhs < lhs);
}
inline bool operator==(Date const & lhs, Date const & rhs) {
  return !(lhs < rhs) && !(rhs < lhs);
}
inline bool operator!=(Date const & lhs, Date const & rhs) {
  return !(lhs == rhs);
}
```

- **Boost provides base classes to implement (inherit) derived operators**
- **Inherit from `boost::less_than_comparable`**
  - `private` inheritance is enough
- **`boost::less_than_comparable`**
  - requires <
  - provides >, <= and >=
- **See boost documentation**
  - www.boost.org

Date.h

```
#include "boost/operators.hpp"
#include <tuple>

class Date : private boost::less_than_comparable<Date> {
  int year, month, day;
public:
  bool operator<(Date const & rhs) const {
    return std::tie(year, month, day) <
           std::tie(rhs.year, rhs.month, rhs.day);
  }
};
```

- **Output operator as free operator:**
  - operator<<
  - Parameters: std::ostream & and Date const &
  - Returns std::ostream & for chaining output
- **Problem: Can only access private members as "friend"**

Any.cpp

```cpp
#include "Date.h"
#include <iostream>

void foo() {
  std::cout << Date::myBirthday;
}
```

Date.h

```cpp
#include <ostream>

class Date {
  int year, month, day; //private ☹

  friend inline std::ostream & operator<<(std::ostream & os, Date const & date) {
    os << date.year << "/" << date.month << "/" << date.day;
    return os;
  }
};
```

Sending Date to std::ostream with print() Member Function

- **Indirection `print` member function**
  - Has access to private data members
  - Can be called from non-friend `operator<<`
  - `operator<<` can be defined in header file

Any.cpp

```
#include "Date.h"
#include <iostream>

void foo() {
  std::cout << Date::myBirthday;
}
```

Date.h

```
#include <ostream>

class Date {
  int year, month, day;
public:
  std::ostream & print(std::ostream & os) const {
    os << year << "/" << month << "/" << day;
    return os;
  }
};
inline std::ostream & operator<<(std::ostream & os, Date const & date) {
  return date.print(os);
}
```

- **Input operator has the same problems as the output operator**
  - operator>>
  - Parameters: std::istream & and Date &
  - Returns std::istream & for chaining input

Any.cpp

```cpp
#include "Date.h"
#include <iostream>

void foo() {
  Date d{};
  std::cin >> d;
}
```

Date.h

```cpp
#include <istream>

class Date {
  int year, month, day;
public:
  std::istream & read(std::istream & is) {
    //Logic for reading values and verifying correctness
    return is;
  }
};
inline std::istream & operator>>(std::istream & is, Date & date) {
  return date.read(is);
}
```

Implementing read() or operator>>()

- **Expect `std::istream` to be in `good()` state as precondition and to provide a correct date**
- **If extracting a date fails set the `std::istream` to fail state**
- **Do not overwrite the this object if the input cannot be used to read a valid date object**
  - Keep the invariant "Date represents a valid date"

Date.h

```
//Includes

class Date {
  int year, month, day;
public:
 std::istream & read(std::istream & is) {
    int year{-1}, month{-1}, day{-1};
    char sep1, sep2;
    //read values
    is >> year >> sep1 >> month >> sep2 >> day;
    try {
      Date input{year, month, day};
      //overwrite content of this object (copy-ctor)
      (*this) = input;
      //clear stream if read was ok
      is.clear();
    } catch (std::out_of_range const & e) {
      //set failbit
      is.setstate(std::ios::failbit);
    }
    return is;
  }
};
```

- **Declaration of a constructor that takes an `std::istream &` as parameter**

```
explicit Date(std::istream & in);
```

- **Declare constructors with one parameter explicit to avoid automatic conversion**
- **Throw an exception if the input does not represent a valid date**
  - For not violating the invariant
  - Alternative: Create a date with a default value

Date.h

```
#ifndef DATE_H_
#define DATE_H_

class Date {
  //...
  explicit Date(std::istream & in);
};

#endif /* DATE_H_ */
```

Date.cpp

```
#include "Date.h"

Date::Date(std::istream & in)
    : year{}, month{}, day{} {
  read(in);
  if (in.fail())
    throw std::out_of_range{"invalid date"};
}
```

Factory Function for Creating Date Objects

- **Declaration of a factory function for Date**

```
Date make_date(std::istream & in);
```

- **Factory functions**
  - `make_xxx()` or `create_xxx()`
- **Placed in**
  - Class as static member function
  - Or in same namespace as the class
- **Delivers a default value if reading fails**
  - E.g. `Date{9999, 12, 31}` or an empty `optional<Date>{}`
  - Similar to `std::string::npos` for "not found"

```
Date make_date(std::istream & in)
try {
  return Date{in};
} catch (std::out_of_range const &) {
  return Date{9999, 12, 31};
}
```

```
std::optional<Date>
make_date(std::istream & in)
try {
  return Date{in};
} catch (std::out_of_range const &) {
  return {};
}
```


```
struct SwissGrid {
  SwissGrid() = default;
  SwissGrid(double y, double x);
  void read(std::istream & in) {
    double y{}; in >> y;
    double x{}; in >> x;
    try {
      SwissGrid inputCoordinate{y, x};
      *this = inputCoordinate;
    } catch(std::invalid_argument & e) {
      in.setstate(std::ios_base::failbit);
    }
  }
private:
  double y{600000.0};
  double x{200000.0};
};
std::istream & operator>>(
                      std::istream & in,
                      SwissGrid & coordinate) {
  coordinate.read(in);
  return in;
}
```

```
struct SwissGrid {
  SwissGrid() = default;
  SwissGrid(double y, double x);
  void read(std::istream & in) {
    double y{}; in >> y;
    double x{}; in >> x;
    try {
      SwissGrid inputCoordinate{y, x};
      *this = inputCoordinate;
    } catch(std::invalid_argument & e) {
      in.setstate(std::ios_base::failbit);
    }
  }
private:
  double y{600000.0};
  double x{200000.0};
};
std::istream & operator>>(
                      std::istream & in,
                      SwissGrid & coordinate) {
  coordinate.read(in);
  return in;
}
```

?

Correct

The input (and output) operator must be implemented as free fuction. Since they usually won't have access to the private members of a type, a member function for reading/writing is required.

Summary

- **Separate the class declaration from the member function implementations properly into header and source files**
- **Initialize member variables with default values or in the constructor's initializer list**
- **Throw an exception from a constructor if it cannot establish the class invariant**
- **You need to implement input and output operators as free functions**
- **Provide sensible operations when implementing operators for your types**